Notice d'utilisation du parallélisme

Résumé :

Toute simulation Code_Aster peut bénéficier des gains de performance que procure le parallélisme. Les gains peuvent être de deux ordres: sur le temps de calcul et sur l'espace RAM/disque requis.

Code_Aster propose sept stratégies parallèles pour s'adapter à l'étude et à la plate-forme de calcul. Certaines sont plutôt axées sur des aspects informatiques (parallélisme informatique), d'autres sont plus algorithmiques (parallélismes numériques) et une troisième catégorie est basée sur une vision multidomaine du problème mécanique (parallélisme multidomaine).

Ce document décrit brièvement l'organisation du parallélisme dans le code. Puis il liste quelques conseils et rappelle quelques fondamentaux pour aider l'utilisateur à tirer parti des stratégies parallèles. Puis on détaille leurs mises en oeuvre, leurs périmètres d'utilisation et leurs gains potentiels. Les cumuls de différentes stratégies (parallélisme hybride ou extension de la portion parallèle du code) sont aussi abordés.
L'utilisateur pressé peut d'emblée se reporter au chapitre 2 (« Le parallélisme en un clic ! »). Il résume le mode opératoire pour mettre en oeuvre la stratégie parallèle préconisée par défaut.

Remarque:

Pour utiliser Code_Aster en parallèle, (au moins) trois cas de figures peuvent se présenter:
on a accès à la machine centralisée Aster et on souhaite utiliser l'interface d'accès Astk,
on effectue des calculs sur un cluster ou sur une machine multicoeur avec Astk,
Idem que le cas précédent mais sans Astk.

Table des Matières

1 Le parallélisme en un clic !.....3
2 Généralités.....4
2.1 Parallélismes informatiques.....4
2.2 Parallélismes numériques 4
2.3 Parallélisme mécanique ou multidomaine 4
3 Quelques conseils préalables.....6
3.1 Préambule.....6
3.2 Calculs indépendants.....6
3.3 Gain en mémoire RAM.....7
3.4 Gain en temps.....7
4 Parallélismes informatiques.....8
4.1 Rampes de calculs indépendants.....8
4.2 Calculs élémentaires et assemblages.....9
4.2.1 Mise en oeuvre.....9
4.2.2 Déséquilibrage de charge.....10
4.2.3 Structures de données distribuées.....11
4.3 Calculs élémentaires d'algèbre linéaire.....12
5 Parallélismes numériques.....14
5.1 Solveur direct MULT_FRONT.....14
5.2 Solveur direct MUMPS.....15
5.3 Solveur itératif PETSC.....17
6 Parallélisme multi-domaine.....18
6.1 Solveur hybride FETI.....18
7 Annexe 1 : Construction d'une version parallèle de Code_Aster.....20
7.1 Version parallèle OpenMP.....20
7.2 Version parallèle MPI.....20

1 Le parallélisme en un clic !

Désormais, la **mise en œuvre du parallélisme dans *Code_Aster* s'effectue de manière** (quasi) **transparente pour l'utilisateur**. Généralement, l'essentiel des coûts en temps et en mémoire du calcul proviennent des constructions et des résolutions de systèmes linéaires.
Il faut donc commencer par repérer ces résolutions dans le fichier de commande (opérateurs STAT_NON_LINE, THER_LINEAIRE...) et modifier leurs paramétrages de manière à utiliser un solveur linéaire parallèle performant. Pour ce faire, on spécifie la valeur 'MUMPS' au mot-clé SOLVEUR/METHODE.

Il reste à préciser le nombre de processeurs souhaités (menu Options d'A*stk*) et la mise en œuvre du parallélisme s'initialise avec un paramétrage par défaut. Sur le serveur centralisé, il faut **paramétrer les champs suivants** :

- mpi_nbcpu=m, nombre de processeurs alloués en MPI.
- (facultatif) mpi_nbnoeud=p, nombre de nœuds sur lesquels vont être dispatchés ces m processus MPI.

Par exemple, si les nœuds réservés au calcul parallèle sont composés de 16 processeurs, afin d'allouer 20 processus MPI à raison de 10 processeurs par nœud, on positionne mpi_nbcpu à 20 et mpi_nbnoeud à 2.

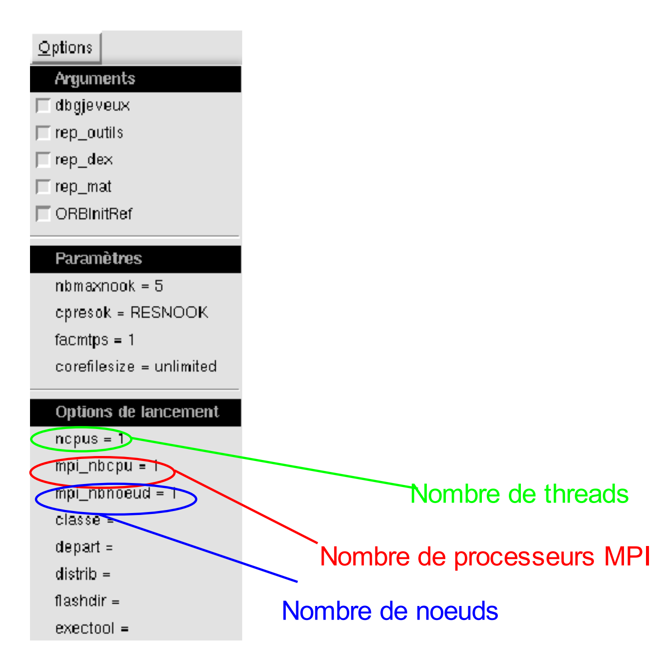


Figure 2.1._ Paramètres d'Astk consacrés au parallélisme

Une fois ce nombre de processus MPI fixé, on peut lancer son calcul (en batch sur la machine centralisé) comme on le ferait en séquentiel. Sauf, qu'avec le parallélisme, on peut bien sûr réduire les spécifications en temps et en mémoire du calcul.

Remarque:

> *Ce type de parallélisme correspond au chaînage des stratégies de calcul parallèle 1b et 2b décrites au paragraphe suivant. Il existe d'autres alternatives qui peuvent être plus efficaces/appropriées suivant les cas de figure (cf. les conseils du § 4).*

2 Généralités

Toute simulation *Code_Aster* peut bénéficier des gains[1] de performance que procure le parallélisme. Du moment qu'il effectue des calculs élémentaires/assemblages, des résolutions de systèmes linéaires ou des simulations indépendantes/similaires. Les gains peuvent être de deux ordres : sur le temps de calcul et sur l'espace RAM/disque requis.
Comme la plupart des codes généralistes en mécanique des structures, *Code_Aster* propose différentes stratégies pour s'adapter à l'étude et à la plate-forme de calcul :

2.1 Parallélismes informatiques

1a/ Lancement de rampes de calculs indépendants/similaires (calculs paramétriques, tests...); Parallélisme via un script shell; Gain en CPU. Lancement standard via Astk.

1b/ Distribution des calculs élémentaires et des assemblages matriciels et vectoriels dans les pré/post-traitements et dans les constructions de système linéaire; Parallélisme MPI; Gain en CPU, et même gain en mémoire avec MUMPS+MATR_DISTRIBUEE et FETI. Lancement standard via Astk.

1c/ Distribution des calculs d'algèbre linéaire effectués par des Blas multithreadées; Gain en CPU; Utilisation avancée.

2.2 Parallélismes numériques

2a/ *Solveur direct* `MULT_FRONT`; Parallélisme OpenMP; Gain en CPU. Lancement standard *via* Astk.

2b/ *Solveur direct* `MUMPS`; Parallélisme MPI; Gain en CPU et en mémoire. Lancement standard *via* Astk.

2c/ *Solveur itératif* `PETSC`; Parallélisme MPI du solveur de Krylov (généralement pas du préconditionneur); Gain en CPU et en mémoire. Lancement standard *via* Astk.

2.3 Parallélisme mécanique ou multidomaine

3a/ *Solveur hybride* `FETI`; Parallélisme MPI; Gain en CPU et en mémoire. Lancement standard *via* Astk.

Les parallélismes numériques 2a et surtout 2b sont les plus plébiscités. Ils supportent une utilisation «industrielle» et «grand public». Ces parallélismes généralistes et fiables procurent des gains notables en CPU et en RAM. Leur paramétrisation est simple, leur mise en œuvre facilitée *via* les menus de l'outil Astk. Ils peuvent facilement se cumuler, en amont et/ou en aval des systèmes linéaires, avec le parallélisme des calculs élémentaires (1b).

Remarque:

D'autres cumuls de parallélisme peuvent s'opérer. Leur mise en oeuvre n'est pas automatisée et ils s'adressent à des utilisateurs plutôt avancés: 1c+2b ou 2c, 2a+3a, 1a+2a/2b/2c/3a, 1a+1c+2b... Certains cumuls sont cependant proscrits car contre-productifs (1c+2a) ou non prévus fonctionnellement (2b/2c+3a). D'autres sont intrinsèques à la méthode (par ex. 1b+3a).

D'un point de vue pratique, l'utilisateur n'a plus à se soucier, pour une utilisation standard, de leur mise en œuvre en interaction fine avec les solveurs linéaires. En renseignant des menus dédiés d'Astk[2], on fixe le nombre de processeurs requis (en terme MPI et/ou OpenMP), avec éventuellement leur répartition par nœud.
Dans le déroulement du fichier de commande *Aster*, si plusieurs processus MPI sont activés (paramètre `mpi_nbcpu` pour 1b/2b/2c/3a), on distribue les mailles entre les processeurs (cf figure

1 Ceux ci sont variables suivant les fonctionnalités sollicitées et leur paramétrage, le jeu de données et la plate-forme logicielle utilisée. Il s'agit essentiellement de gain en temps de calcul. Sauf dans le cas de `MUMPS`, de `PETSC` et de `FETI`, où l'on peut aussi gagner en mémoire.

2 Menus `Options+ncpus/mpi_nbcpus/mpi_nbnoeud`.

3.1). Cette distribution se décline de différentes manières et elle est paramétrable dans les opérateurs `AFFE/MODI_MODELE`.

Dans le cas d'opérateurs de calcul (`MECA_STATIQUE`, `STAT_NON_LINE`...), cette distribution des mailles des calculs élémentaires va ensuite nourrir le parallélisme «numérique» des solveurs linéaires choisis par le mot-clé `SOLVEUR`.
Suivant le cas de figure, **solveur linéaire distribué** (`MUMPS`, `FETI`) **ou non** (`MULT_FRONT`, `PETSC`), on fournit les données, par morceaux, ou après les avoir rassemblées, au solveur linéaire. Ce dernier souvent les redistribue en interne au grès de ses contingences. Mais dans le cas d'un solveur linéaire acceptant un flot de données d'entrée déjà distribué, les gains en temps et en mémoire sont patents.
Par ailleurs, on ne **rassemble les données** (ou on effectue les communications MPI idoines) que si le déroulement algorithmique l'impose : par exemple pour un calcul de maximum sur toutes les mailles du modèle ou pour un produit matrice-vecteur.
Si le **solveur linéaire parallèle utilise des threads** (OpenMP des scénarios 1c et 2a) ceux-ci s'exécutent indépendamment des éventuels aspects MPI. Dans le cas le plus usuel (2a), les threads se substituent ensuite aux processus MPI. Dans un cas de parallélisme hybride (1c+2b/c), ils démultiplient leur efficacité.

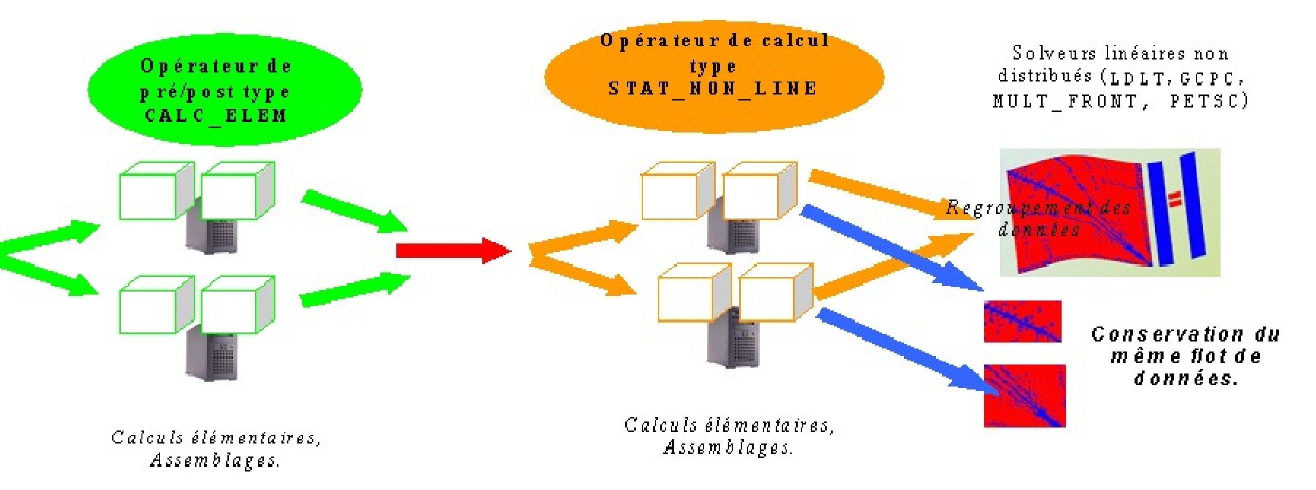


Figure 3.1._ Lors d'un calcul paralléle, distribution des flots de données/traitements suivant les opérateurs (pré/post-traitement ou résolution) et le type de solveur utilisé.

Dans le cas d'un **opérateur de pré/post-traitements** (type `CALC_ELEM`), là encore, on ne rassemble les données (ou on effectue les communications MPI idoines) que si le déroulement algorithmique l'impose.
De toute façon, **tout calcul parallèle *Aster*** (sauf bien sûr le scénario 1a de calculs indépendants) **doit respecter le paradigme suivant** : les bases globales de chaque processeur sont identiques à la fin de chaque opérateur. Car on ne sait pas si l'opérateur qui suit dans le fichier de commandes a prévu un flot de données incomplet. Il faut donc organiser les communications idoines pour compléter les champs éventuellement incomplets.

Remarque:

Entre chaque commande, l'utilisateur peut même changer la répartition des mailles suivant les processeurs. Il lui suffit d'utiliser la commande `MODI_MODELE` . Ce mécanisme reste licite lorsqu'on enchaîne différents calculs (mode `POURSUITE`). La règle étant bien sûr que cette nouvelle répartition perdure, pour le modèle concerné, jusqu'à l'éventuel prochain `MODI_MODELE` et que cette répartition doit rester compatible avec le paramétrage parallèle du calcul (nombre de nœuds/processeurs...).

3 Quelques conseils préalables

On formule ici **quelques conseils pour aider l'utilisateur à tirer parti des stratégies de calcul parallèle du code**. Mais il faut bien être conscient, qu'avant tout chose, il faut d'abord optimiser et valider son calcul séquentiel en tenant compte des conseils qui fourmillent dans les documentations des commandes. Pour ce faire, on utilise, si possible, un maillage plus grossier et/ou on n'active que quelques pas de temps.

Le paramétrage par défaut et les affichages/alarmes du code proposent un fonctionnement équilibré et instrumenté. On liste ci-dessous et, de manière non exhaustive, plusieurs questions qu'il est intéressant de se poser lorsqu'on cherche à paralléliser son calcul. Bien sûr, certaines questions (et réponses) sont cumulatives et peuvent donc s'appliquer simultanément.

3.1 Préambule

Il est intéressant de **valider**, au préalable, **son calcul parallèle** en comparant quelques itérations en mode séquentiel et en mode parallèle. Cette démarche permet aussi de **calibrer les gains maximums atteignables** (speed-up théoriques) et donc d'éviter de trop «gaspiller de processeurs». Ainsi, si on note f la portion parallèle du code (déterminée par exemple *via* un run séquentiel préalable), alors le speed-up théorique $\mathbf{S_p}$ maximal accessible sur p processeurs se calcule suivant la formule d'Amdhal (cf. [R6.01.03] §2.4) :

$$S_p=\frac{1}{1-f+\frac{f}{p}}$$

Par exemple, si on utilise le parallélisme `MUMPS` distribué par défaut (scénarios 1b+2b) et que les étapes de construction/résolution de système linéaire représentent 90% du temps séquentiel (f=0.90), le speed-up théorique est borné à la valeur $\mathbf{S}_\infty=\frac{1}{1-0.9+0.9/\infty}=10$! Et ce, quelque soit le nombre de processus MPI alloués.

Il est intéressant d'**évaluer les principaux postes de consommation (temps/RAM) :** en mécanique quasi-statique, ce sont généralement les étapes de **calculs élémentaires/assemblages**, de **résolution de système linéaire** et les algorithmes de **contact-frottement**. Mais leurs proportions dépendent beaucoup du cas de figure (caractéristiques du contact, taille du problème, complexité des lois matériaux...). Si les calculs élémentaires sont importants, il faut les paralléliser *via* la scénario 1b (scénario par défaut). Si, au contraire, les systèmes linéaires focalisent l'essentiel des coûts, les scénarios 2a ou 2b peuvent suffire. Par contre, si c'est le contact-frottement qui dimensionne le calcul, il faut chercher à optimiser son paramétrage et/ou paralléliser son solveur linéaire interne (cf. méthode `GCP+MUMPS`).

Pour optimiser son calcul parallèle, il faut surveiller les éventuels **déséquilibres de charge** du flot de données (CPU et mémoire) et limiter les surcoûts dus **aux déchargements mémoire** (`JEVEUX` et `MUMPS` OOC) et aux **archivages de champs**. Sur le sujet, on pourra consulter la documentation [U1.03.03] «Indicateur de performance d'un calcul (temps/mémoire)». Elle indique la marche à suivre pour établir les bons diagnostics et elle propose des solutions.

Quelques chiffres empiriques: on conseille d'allouer au moins 20 000 degrés de liberté par processus MPI; Un calcul thermo-mécanique standard bénéficie généralement, sur 32 processeurs, d'un gain de l'ordre de la dizaine en temps *elapsed* et d'un facteur 4 en mémoire RAM.

3.2 Calculs indépendants

Lorsque la simulation que l'on souhaite effectuer se décompose naturellement (étude paramétrique, calcul de sensibilité...) ou, artificiellement (chaînage thermo-mécanique particulier...), en calculs similaires mais indépendants, on peut gagner beaucoup en temps calcul grâce au parallélisme numérique 1a.

3.3 Gain en mémoire RAM

Lorsque le facteur mémoire dimensionnant concerne la résolution de systèmes linéaires (ce qui est souvent le cas), le cumul des parallélismes informatiques 1b (calculs élémentaires/assemblages) et numériques 2b (solveur linéaire distribué MUMPS) est tout indiqué[3].

Une fois que l'on a distribué son calcul *Code_Aster*+MUMPS sur suffisamment de processeurs, les consommations RAM de JEVEUX peuvent devenir prédominantes (par rapport à celles de MUMPS que l'on a étalées sur les processeurs). Pour rétablir la bonne hiérarchie (le solveur externe doit supporter le pic de consommation RAM !) il faut activer, en plus, l'option SOLVEUR/MATR_DISTRIBUEE [U4.50.01].

Pour résoudre des problèmes frontières de très grandes tailles ($>5.10^6$ degrés de liberté), on peut aussi essayer les solveurs itératifs/hybride parallèles (stratégies parallèles 2c/3a).

3.4 Gain en temps

Si l'essentiel des coûts concerne uniquement les résolutions de systèmes linéaires on peut se contenter d'utiliser les solveurs linéaires MUMPS en mode centralisé (stratégie 2b) ou MULT_FRONT (2a). Dès que la construction des systèmes devient non négligeable (>5%), il est primordial d'étendre le périmètre parallèle en activant la distribution des calculs élémentaires/assemblages (1b) et en passant à MUMPS distribué (valeur par défaut).

Sur des problèmes frontières de grandes tailles ($>10^6$ degrés de liberté), une fois les bons paramètres numériques sélectionnés[4] (préconditionneur, relaxation... cf. [U4.50.01]), les solveurs itératifs/hybride parallèles (2c/3a) peuvent procurer des gains en temps très appréciables par rapport aux solveurs directs génériques (2a/2b). Surtout si les résolutions sont relaxées[5] car, par la suite, elles sont corrigées par un processus englobant (algorithme de Newton de THER/STAT_NON_LINE...).

3 Pour baisser les consommations mémoire de MUMPS on peut aussi jouer sur d'autres paramètres : OOC et relaxation des résolutions [U4.50.01].
4 Il n'y a par contre pas de règle universelle, tous les paramètres doivent être ajustés au cas par cas.
5 C'est-à-dire que l'on va être moins exigeant quant à la qualité de la solution. Cela peut passer par un critère d'arrêt médiocre, le calcul d'un préconditionneur frustre et sa mutualisation durant plusieurs résolutions de système linéaire... Les fonctionnalités des solveurs non linéaires et linéaires de *Code_Aster* permettent de mettre en œuvre facilement ce type de scénarios.

4 Parallélismes informatiques

4.1 Rampes de calculs indépendants

Utilisation : grand public *via Astk*.
Périmètre d'utilisation : calculs indépendants (paramétrique, étude de sensibilité...).
Nombre de processeurs conseillés : limite de la machine/gestionnaire *batch*.
Gain : en temps CPU.
Speed-up : proportionnel au nombre de cas indépendants.
Type de parallélisme : script shell.
Scénario : 1a du §3. Cumul avec toutes les autres stratégies de parallélisme licite mais s'adressant à des utilisateurs avancés (hors périmètre d'*Astk*).

L'outil ***Astk*** **permet d'effectuer toute une série de calculs similaires mais indépendants** (en séquentiel et surtout en parallèle MPI). On peut utiliser une version officielle du code ou une surcharge privée préalablement construite. Les fichiers de commande explicitant **les calculs sont construits dynamiquement à partir d'un fichier de commande «modèle» et d'un mécanisme de type «dictionnaire» :** jeux de paramètres différents pour chaque étude (mot-clé `VALE` du fichier `.distr`), blocs de commandes *Aster*/Python variables (`PRE/POST_CALCUL`)...
Le lancement de ces rampes parallèles s'effectue avec les paramètres de soumission usuels d'*Astk*. On peut même re-paramétrer la configuration matérielle du calcul (liste des nœuds, nombre de cœurs, mémoire RAM totale par nœud...) *via* un classique fichier `.hostfile`.

Pour plus d'informations sur la mise en oeuvre de ce parallélisme, on pourra consulter la documentation `d'Astk` U1.04.00.

Remarques :

- *Avant de lancer une telle rampe de calculs, il est préférable d'optimiser au préalable sur une étude type, les différents paramètres dimensionnants : gestion mémoire `JEVEUX`, aspects solveurs non linéaire/modaux/linéaire, mot-clé `ARCHIVAGE`, algorithme de contact-frottement... (cf. doc. U1.03.03 sur les performances, U4.50.01...).*
- *On peut facilement écrouler une machine en lançant trop de calculs vis-à-vis des ressources disponibles. Il est conseillé de procéder par étapes et de se renseigner quant aux possibilités d'utilisations de moyens de calculs partagés (classe batch, gros jobs prioritaires...).*

4.2 Calculs élémentaires et assemblages

Utilisation : grand public *via Astk*.
Périmètre d'utilisation : calculs comportant des calculs élémentaires/assemblages coûteux (mécanique non linéaire).
Nombre de processeurs conseillés : 4 ou 8. Couplé avec le parallélisme distribué de MUMPS (valeur par défaut), typiquement 16 ou 32.
Gain : en temps voire en mémoire avec solveur linéaire MUMPS (si MATR_DISTRIBUEE) ou FETI.
Speed-up : Gains variables suivant les cas (efficacité parallèle[6]>50%). Il faut une assez grosse granularité pour que ce parallélisme reste efficace : 5.10^4 degrés de liberté par processeur.
Type de parallélisme : MPI.
Scénario: 1b du §3. Nativement conçu pour se coupler aux parallélismes numériques 2b (*via Astk*). Cumul avec toutes les autres stratégies de parallélisme licite. Cumul 1b+3a intrinsèque.

4.2.1 Mise en oeuvre

Désormais, la mise en œuvre du parallélisme dans les opérateurs effectuant des calculs élémentaires/ assemblages s'effectue de manière transparente à l'utilisateur. Elle s'initialise par défaut dès qu'on lance un calcul *via Astk* (menu `Options`) utilisant plusieurs processeurs.
Ainsi sur le serveur centralisé *Aster*, **il faut paramétrer les champs suivants :**

- `mpi_nbcpu=m`, nombre de processeurs alloués en MPI (nombre de processus MPI).
- (facultatif) `mpi_nbnoeud=p`, nombre de noeuds sur lesquels vont être dispatchés ces processus MPI.

Par exemple, sur la machine *Aster* actuelle, les nœuds sont composés de 16 processeurs. Pour allouer 20 processus MPI à raison de 10 processus par nœud, il faut donc positionner `mpi_nbcpu` à `20` et `mpi_nbnoeud` à `2`.
Une fois ce nombre de processus MPI fixé, on peut lancer son calcul (en *batch* sur la machine centralisé) avec le même paramétrage qu'en séquentiel. On peut bien sûr baisser les spécifications en temps et en consommations mémoire (JEVEUX) du calcul (si MUMPS+MATR_DISTRIBUEE ou FETI) par rapport aux consommations séquentielles.

Dès que plusieurs processus MPI sont activés, l'affectation du modèle dans le fichier de commandes *Aster* (opérateur AFFE_MODELE) **distribue les mailles[7] entre les processeurs**. *Code_Aster* étant un code éléments finis, c'est la distribution naturelle des données (et des tâches associées). Par la suite, les étapes *Aster* de calculs élémentaires et d'assemblages (matriciels et vectoriels) vont se baser sur cette distribution pour «tarir» les flots de données/traitements locaux à chaque processeur. Chaque processeur ne va effectuer que les calculs associés au groupe de maille dont il a la charge.

Cette **répartition maille/processeur** se décline de différentes manières et elle est paramétrable dans l'opérateur AFFE_MODELE[U4.41.01] *via* les valeurs du mot-clé PARALLELISME:

1) CENTRALISE: **Les mailles ne sont pas distribuées** (comme en séquentiel). Chaque processeur connaît tout le maillage. Le parallélisme 1b n'est donc pas mis en œuvre. Ce mode d'utilisation est utile pour les tests de non-régression et pour certaines études où le parallélisme 1b rapporte peu voire est contre-productif (par ex. si on doit rassembler les données élémentaires pour nourrir un système linéaire non distribué et que les communications MPI requises sont trop lentes). Dans tous les cas de figure où les calculs élémentaires représentent une faible part du temps total (par ex. en élasticité linéaire), cette option peut être suffisante.
2) GROUP_ELEM**(défaut)/**MAIL_DISPERSE/MAIL_CONTIGU/SOUS_DOMAINE : les mailles sont distribuées suivant le **type de mailles**, une **distribution cyclique**, par **paquets de mailles contiguës** ou suivant des **sous-domaines** (préalablement construits *via* un des opérateurs DEFI_PART_FETI/DEFI_PART_OPS).

6 On gagne au moins un facteur 2 (sur les temps consommés par les étapes parallélisées) en quadruplant le nombre de processeurs.
7 Du modèle.

Remarques :

- *De manière générale, la distribution par sous-domaine (PARALLELISME=SOUS_DOMAINE + PARTITION) est plus souple et elle peut s'avérer plus efficace en permettant d'adapter plus finement le flot de données à sa simulation.*
- *Entre chaque commande, l'utilisateur peut même changer la répartition des mailles suivant les processeurs. Il lui suffit d'utiliser la commande MODI_MODELE . Ce mécanisme reste licite lorsqu'on enchaîne différents calculs (mode POURSUITE). La règle étant bien sûr que cette nouvelle répartition reste valable, pour le modèle concerné, jusqu'à l'éventuel prochain MODI_MODELE et que cette répartition doit rester compatible avec le paramétrage parallèle du calcul (nombre de nœuds/processeurs...).*
- *Pour l'instant, la distribution par sous-domaines est basée sur un partitionnement au sens FETI via les opérateurs DEFI_PART_*** . Ces derniers génèrent en fait une structure de données trop riches pour les seuls besoins de la distribution maille/processeur (description interface, découpage des charges/conditions limites...). Pour éviter les surcoûts calculs qu'implique la constitution de ces objets redondants, on peut juste se contenter de préciser une seule charge, voire une charge «bidon». Pourvu qu'elle n'implique aucune manipulation de Lagrange (par exemple une charge de pression). Cette information ne servira pas dans le calcul effectif des seuls objets décrivant la distribution maille/processeur.*

4.2.2 Déséquilibrage de charge

Pour optimiser un calcul parallèle, il faut essayer d'équilibrer le flot de traitements qui va être confié à chaque processeur. Pour ce faire, ne disposant pas d'heuristique générale pour estimer la charge calcul en fonction des données, on ne peut que conseiller de distribuer de manière homogène les données entre les processeurs, tout en respectant deux règles spécifiques aux modélisations du code :

- **Rester attentif aux mailles de peau.**
Les distributions par mailles sont les plus simples et les plus robustes à mettre en œuvre mais elles peuvent conduire à des déséquilibres de charge car elles ne tiennent pas compte des mailles particulières (mailles de peau, mailles spectatrices, zones non linéaires...). Par exemple, avec un fichier maillage débutant par 100 mailles 2D puis 900 mailles 3D, une répartition en 10 paquets *via* MAIL_CONTIGU va conduire à un déséquilibrage de charge : le premier processeur effectue des calculs élémentaires/assemblages sur des mailles 2D alors que les 9 autres ne travaillent qu'en 3D. Dans ce cas de figure, la distribution cyclique[8] ou celle par sous-domaines[9] paraissent plus adaptées.

- **Rester attentif à la proportion de Lagranges.**
Une autre cause de déséquilibrage de charge entre processeurs, peut provenir des **conditions de Dirichlet par dualisation** (DDL_IMPO, LIAISON_***...). Chacun de ces blocages introduit une nouvelle variable (dite tardive), un Lagrange, et un nouvel élément fini (dit tardif). Par soucis de robustesse/maintenabilité/lisibilité, le traitement de ces mailles tardives est affecté uniquement au processeur maître. Cette surcharge de travail est souvent négligeable (cf. §3 doc. U1.03.03 sur les performances), mais dans certain cas, elle peut introduire un déséquilibre plus marqué. L'utilisateur peut le compenser en renseignant un des mot-clés CHARGE_PROC0_MA/SD (MA pour maille et SD pour sous-domaine) de AFFE_MODELE[U4.41.01].
Par exemple, en posant CHARGE_PROC0_MA=50 (%), le processeur maître pourra plus se consacrer aux mailles tardives car il n'aura que 50% des mailles standards (dites physiques) qu'il aurait dû avoir. Le reliquat est distribué entre les autres processeurs. De même avec une répartition par sous-domaines, si CHARGE_PROC0_SD=1 (sd), le processeur 0 ne traite qu'un sous-domaine, le reliquat est sous-traité.

Remarques :

- *Les paramètres CHARGE_PROC0_MA/SD peuvent être modifiés en cours de calcul via l'opérateur MODI_MODELE . On peut aussi utiliser ce mécanisme pour artificiellement*

8 Maille 1 pour le proc. 1, maille 2 pour le proc. 2... maille 10 pour le proc. 10 puis on revient au premier processeur, maille 11/proc. 1 etc.
9 Si on partitionne en 10 sous-domaines, chacun va comporter 100 mailles 3D auxquelles vont être rattachées 10 mailles de peau 2D.

surcharger le processeur maître (si par exemple, il est «mappé» sur un nœud de calcul disposant de plus de mémoire ou plus rapide).

- *Le traitement différencié des mailles tardives concernent principalement les Lagranges, mais pas seulement. Ce cas de figure se retrouvent aussi avec les forces nodales, les échanges thermiques, la méthode de contact continue...*

4.2.3 Structures de données distribuées

La distribution des données qu'implique ce type de parallélisme numérique ne diminue pas forcément les consommations mémoire JEVEUX. Par soucis de lisibilité/maintenabilité, les objets *Aster* usuels sont initialisés avec la même taille qu'en séquentiel. On se «contente» juste de les remplir partiellement suivant les mailles dont a la charge le processeur. On ne retaille donc généralement pas ces structures de données au plus juste, elles comportent beaucoup de valeurs nulles.
Cette stratégie n'est tenable que tant que les objets JEVEUX principaux impliqués dans les calculs élémentaires/assemblages (CHAM_ELEM, RESU_ELEM, MATR_ASSE et CHAM_NO) ne dimensionnent pas les contraintes mémoire du calcul (cf. §5 de [U1.03.03]). Normalement leur occupation mémoire est négligeable comparée à celle du solveur linéaire. Mais lorsque ce dernier (par ex. MUMPS) est lui aussi Out-Of-Core[10] et parallélisé en MPI (avec la répartition des données entre processeurs que cela implique), cette hiérarchie n'est plus forcément respectée. D'où l'introduction d'**une option** (mot-clé MATR_DISTRIBUEE cf. U4.50.01) **permettant de véritablement retailler, au plus juste, le bloc de matrice *Aster* propre à chaque processeur.**

Remarques :

- *Pour plus d'informations sur l'impact du parallélisme sur les structures de données et sur les structures de données distribuées, on pourra consulter le wiki du site.*
- *En mode distribué, chaque processeur ne manipule que des matrices incomplètes (retaillées ou non). Par contre, afin d'éviter de nombreuses communications MPI (lors de l'évaluation des critères d'arrêt, calculs de résidus...), ce scénario n'a pas été retenu pour les vecteurs seconds membres. Leurs construction est bien parallélisée, mais à l'issu de l'assemblage, les contributions partielles de chaque processeur sont sommées. Ainsi, tout processeur connaît entièrement les vecteurs impliqués dans le calcul.*

10 L'Out-Of-Core (OOC) est un mode de gestion de la mémoire qui consiste à décharger sur disque certains objets alloués par le code pour libérer de la RAM. La stratégie OOC permet de traiter des problèmes plus gros mais ces accès disque ralentissent le calcul. *A contrario*, le mode In-Core (IC) consiste à garder les objets en RAM. Cela limite la taille des problèmes accessibles mais privilégie la vitesse.

4.3 Calculs élémentaires d'algèbre linéaire

Utilisation : avancée (*Astk* + modification du fichier *.btc)*
Périmètre d'utilisation : calculs comportant des résolutions de systèmes linéaires (*via* MUMPS/PETSC) et/ou des opérations de type BLAS[11] coûteuses.
Nombre de processeurs conseillés : 2 ou 4.
Gain : en temps *elapsed*.
Speed-up : Gains variables suivant les cas (efficacité parallèle[12]>50%). Une granularité faible suffit pour que ce parallélisme reste efficace : 10^4 degrés de liberté par processeur.
Type de parallélisme: threads.
Scénario: 1c du §3. Une utilisation classique consiste à tirer parti d'un parallélisme hybride MPI+threads pour prolonger les performances du MPI (1c+2b...). Cumul contre-productif avec 2a.

Pour initier des traitements parallèles dans les calculs d'algèbre linéaire qui sont utilisés[13] dans le code (routines BLAS), il faut :

- **Préparer son étude** (fichiers commande/maillage...) et **son paramétrage de lancement** (temps, mémoire, nombre de processus Threads/MPI/Nœuds...). Par exemple, sur le serveur centralisé *Aster*, il faut paramétrer les champs suivants :
  - ncpus=n, nombre de processeurs utilisés par les threads.
  - mpi_nbcpu=m, nombre de processeurs utilisés par MPI (nombre de processus MPI).
  - mpi_nbnoeud=p, nombre de nœuds sur lesquels vont être dispatchés ces processus.

  Si (ncpus,mpi_nbcpu)=(n,1), les bibliothèques BLAS threadées utiliseront n processeurs. Par contre, si (ncpus,mpi_nbcpu)=(n,m), chacun des m processus MPI utilisera pour ses besoins en BLAS, n processeurs.
  Au final, *Astk* allouera nxm processeurs. Ceux-ci doivent donc être disponibles sur les p nœuds alloués.

- **Préparer la récupération du script de lancement** généré par *Astk* (mettre en donnée un nouveau fichier mon_script.btc).

- **Pré-lancer l'étude** (elle se termine en quelque secondes juste après l'écriture du .btc; le calcul n'est pas lancé).

- **Modifier «à la main» le contenu de ce fichier** en remplaçant le nom de l'exécutable de lancement *Aster* (/chemin/exec_aster) par un autre nom (/home/user/exec_aster_blasthreadees).

- **Créer ce nouvel exécutable par recopie de l'exécutable officiel.** Modifier le «à la main» afin de permettre l'usage des BLAS threadées. Par exemple, sur la machine centralisée, on positionnera la variable d'environnement MKL_SERIAL à NO plutôt que le YES par défaut: export MKL_SERIAL=NO.

- **Lancer** (pour de bon cette fois) l'étude en prenant soin de ne pas modifier[14] le paramétrage d'*Astk* sauf à mettre le .btc en donnée.

-

Remarques :

- *La mise en oeuvre de ce parallélisme dépend du contexte informatique (matériel, logiciel) et des librairies d'algèbre linéaire threadées que l'on souhaite coupler au processus. On donne ici un canevas dans le cas très précis de l'utilisation des BLAS MKL sur la machine centralisée Aster.*

11 Les BLAS sont des routines optimisées effectuant des opérations d'algèbre linéaire basique (d'où l'acronyme 'Basic Linear Algebra Subroutine') en stockage dense : produit matrice-vecteur, produit scalaire... Elles peuvent être parallélisées en mémoire partagée (OpenMP ou directement des threads Posix).
12 On gagne au moins un facteur 2 (sur les temps consommés par les étapes parallélisées) en quadruplant le nombre de processeurs.
13 Les BLAS sont massivement utilisées dans les solveurs linéaires MUMPS/PETSC.
14 Il deviendrait alors contradictoire à celui sauvegardé dans le .btc

- *Ce type de parallélisme permet de prolonger le parallélisme MPI de* `MUMPS` *lorsqu'on a atteint une trop faible granularité de travail par processus MPI. Le fait d'utiliser 2 ou 4 threads par processus MPI permet de prolonger notablement la scalabilité en temps du calcul. Par contre, on ne gagnera pas plus en mémoire RAM.*

5 Parallélismes numériques

5.1 Solveur direct MULT_FRONT

Utilisation : grand public *via Astk*.
Périmètre d'utilisation : calculs comportant des résolutions de systèmes linéaires coûteuses (en général STAT/DYNA_NON_LINE, MECA_STATIQUE...).
Nombre de processeurs conseillés : 2 ou 4.
Gain : en temps CPU.
Speed-up : Gains variables suivant les cas (efficacité parallèle≈50%). Il faut une grosse granularité pour que ce parallélisme reste efficace : 10^5 degrés de liberté par processeur.
Type de parallélisme : OpenMP.
Scénario: 2a du §3. Cumul contre-productif avec 1c. Chaînage 1b+2a possible et réalisable *via Astk*. Chaînages 1a+2a/2a+3a potentiels (utilisateurs avancés).

Cette méthode multifrontale développée en interne (cf. [R6.02.02] ou [U4.50.01] §3.5) est utilisée *via* la mot-clé SOLVEUR/METHODE='MULT_FRONT'. **C'est le solveur linéaire** (historique et auto-portant) **préconisé par défaut en séquentiel**.
La mise en oeuvre de ce parallélisme s'effectue de manière transparente à l'utilisateur. Elle s'initialise par défaut dès qu'on lance un calcul *via Astk* (menu Options) utilisant plusieurs *threads*.
Ainsi sur le serveur centralisé *Aster*, il faut paramétrer le champs suivants :

- ncpus=n, nombre de processeurs utilisés par les *threads*.

Une fois ce nombre de threads fixé on peut lancer son calcul (en *batch* sur la machine centralisé) avec le même paramétrage qu'en séquentiel. On peut bien sûr baisser les spécifications en temps du calcul.

5.2 Solveur direct MUMPS

Utilisation : grand public *via Astk*.
Périmètre d'utilisation : calculs comportant des résolutions de systèmes linéaires coûteuses (en général STAT/DYNA_NON_LINE, MECA_STATIQUE...).
Nombre de processeurs conseillés : 16 ou 32, voire plus (surtout si couplée avec 1c).
Gain : en temps CPU et en mémoire RAM (y compris la partie JEVEUX si MATR_DISTRIBUEE activée).
Speed-up : Gains variables suivant les cas (efficacité parallèle≈30%). Il faut une granularité moyenne pour que ce parallélisme reste efficace : environ 3.10^4 degrés de liberté par processeur.
Type de parallélisme : MPI.
Scénario : 2b du §3. Nativement conçu pour se coupler aux parallélismes informatiques 1c, et surtout, 1b (*via Astk*). Chaînage 1a+2a potentiel (utilisateurs avancés).

Cette méthode multifrontale s'appuie sur le produit externe MUMPS (cf. [R6.02.03] ou [U4.50.01] §3.7) est utilisée *via* la mot-clé SOLVEUR/METHODE='MUMPS'. **C'est le solveur linéaire conseillé pour exploiter pleinement les gains CPU/RAM que peut procurer le parallélisme**. Ce type de parallélisme est performant (surtout lorsqu'il est couplé avec 1b) tout en restant générique, robuste et grand public.
La mise en œuvre de ce parallélisme s'effectue de manière transparente à l'utilisateur. Elle s'initialise par défaut dès qu'on lance un calcul *via Astk* (menu Options) utilisant plusieurs processus MPI.
Ainsi sur le serveur centralisé *Aster*, **il faut paramétrer les champs suivants** :

- mpi_nbcpu=m, nombre de processeurs alloués en MPI (nombre de processus MPI).
- (facultatif) mpi_nbnoeud=p, nombre de noeuds sur lesquels vont être dispatchés ces processus MPI.

Une fois ce nombre de processus MPI fixé on peut lancer son calcul (en *batch* sur la machine centralisé) avec le même paramétrage qu'en séquentiel. On peut bien sûr baisser les spécifications en temps et en consommations mémoire (JEVEUX/RAM totale) du calcul.

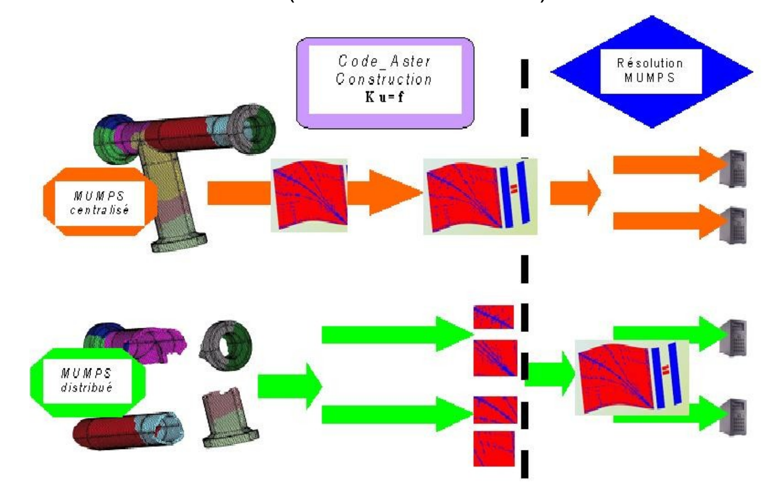


Figure 6.1._ Flots de données/traitements parallèles du couplage Aster+MUMPS suivant le mode d'utilisation: centralisé ou distribué.

Idéalement, ce solveur linéaire parallèle doit être utilisé en mode distribué (PARALLELISME=GROUP_ELEM/MAIL_DISPERSE/MAIL_CONTIGU/SOUS_DOMAINE). C'est-à-dire qu'il faut avoir initié en amont du solveur, au sein des calculs élémentaires/assemblages, un flot de données distribué (scénario parallèle 1b). MUMPS accepte en entrée ces données incomplètes et il les rassemble en interne. On ne perd pas ainsi de temps (comme c'est le cas pour les autres solveurs linéaires) à compléter les données issues de chaque processeur. Ce mode de fonctionnement est activé par défaut dans les commandes AFFE/MODI_MODELE (cf. §5.2).
En mode centralisé (CENTRALISE), la phase amont de construction de système linéaire n'est pas parallélisée (chaque processeur procède comme en séquentiel). MUMPS ne tient compte que des données issues du processeur maître.

Dans le premier cas, le code est parallèle de la construction du système linéaire jusqu'à sa résolution (chaînage des parallélismes 1b+2b), dans le second cas, on n'exploite le parallélisme MPI que sur la partie résolution (parallélisme 2b).

Remarque :

Lorsque la part du calcul consacrée à la construction du système linéaire est faible (<5%), les deux modes d'utilisation (centralisé ou distribué) affichent des gains en temps similaires. Par contre, seule l'approche distribuée procure, en plus, des gains sur les consommations RAM.

5.3 Solveur itératif `PETSC`

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs comportant des résolutions de systèmes linéaires coûteuses (en général `STAT/DYNA_NON_LINE`, `MECA_STATIQUE`...). Plutôt des problèmes bien conditionnés de grandes tailles.
Nombre de processeurs conseillés: quelques dizaines voire centaines.
Gain: en temps CPU et en mémoire RAM (suivant les préconditionneurs).
Speed-up: gains variables suivant les cas (efficacité parallèle>50%). Il faut une granularité moyenne pour que ce parallélisme reste efficace : 3.10^4 degrés de liberté par processeur.
Type de parallélisme: MPI.
Scénario: 2c du §3. Chaînage 1b+2c possible et réalisable *via Astk*. Chaînages 1a+2c/1c+2c potentiels (utilisateurs avancés).

Cette bibliothèque de solveurs itératifs (cf. [R6.01.02] ou [U4.50.01] §3.9) est utilisée *via* la mot-clé `SOLVEUR/METHODE='PETSC'`. **Ce type de solveur linéaire est conseillé pour traiter des problèmes frontières de très grande taille (>5.10^6 degrés de liberté) et plutôt bien conditionnés.**
La mise en œuvre de ce parallélisme s'effectue de manière transparente à l'utilisateur. Elle s'initialise par défaut dès qu'on lance un calcul *via Astk* (menu `Options`) utilisant plusieurs processus MPI.
Ainsi sur le serveur centralisé *Aster*, il faut **paramétrer les champs suivants :**

- `mpi_nbcpu=m`, nombre de processeurs alloués en MPI (nombre de processus MPI).
- (facultatif) `mpi_nbnoeud=p`, nombre de nœuds sur lesquels vont être dispatchés ces processus MPI.

Une fois ce nombre de processus MPI fixé, on peut lancer son calcul (en batch sur la machine centralisé) avec le même paramétrage qu'en séquentiel. On peut bien sûr baisser les spécifications en temps du calcul (voire en mémoire suivant les préconditionneurs).

Remarques :

- *Si le système matriciel que `PETSC` doit résoudre est issu de calculs élémentaires/assemblages parallélisés (stratégie parallèle 2b), ces flots de données incomplets sont automatiquement rassemblés. Pour l'instant, seul un flot de donné centralisé (le même pour tous les processeurs) a été mis en place. Le parallélisme reste interne à `PETSC`. On perd donc, dans le couplage 1b+2c, du temps dans des communications MPI collectives.*
- *Contrairement aux solveurs parallèles directs (`MUMPS` , `MULT_FRONT`), les itératifs ne sont pas universels et robustes. Ils peuvent être très compétitifs (en temps et surtout en mémoire), mais il faut trouver le point de fonctionnement (algorithme, préconditionneur...) adapté au problème.*

6 Parallélisme multi-domaine

6.1 Solveur hybride FETI

Utilisation: grand public *via Astk*.
Périmètre d'utilisation: calculs comportant des résolutions de systèmes linéaires coûteuses (en général STAT/DYNA_NON_LINE, MECA_STATIQUE...). Pour des problèmes frontières. Solveur de recherche. À utiliser plutôt en linéaire.
Nombre de processeurs conseillés: 16 ou 32.
Gain: en temps CPU et en mémoire RAM (surtout).
Speed-up: gains variables suivant les cas. Il faut une assez grosse granularité pour que ce parallélisme reste efficace : 5.10^4 degrés de liberté par processeur.
Type de parallélisme: MPI.
Scénario: 3a du §3. Chaînage 1b+3a intrinsèque. Cumuls 2a+3a/1c+3a possibles *via Astk* (utilisations avancées).

Ce solveur hybride par Décomposition de Domaines (cf. [R6.01.03] ou [U4.50.01] §3.10) est utilisée *via* la mot-clé SOLVEUR/METHODE='FETI'. **Ce type de solveur linéaire peut être utilisé pour traiter des problèmes frontières de très grande taille (>5.10^6 degrés de liberté) ou lorsqu'on souhaite réaliser des gains mémoire importants**[15].
Comme pour le solveur MUMPS en mode distribué, cette méthode par DD initie un flot de données parallèle dès la construction du système linéaire (calculs élémentaires/assemblages). C'est pour cette raison que l'on stipule que le chaînage 1b+3a est intrinsèque à la méthode. On ne peut pas les séparer. Chaque processeur va être responsable d'un certain nombre de sous-domaines et il va assembler leurs données locales (matrices, seconds membres). À charge, par la suite, au solveur d'interface géré par le processeur maître de coordonner ces contributions *via* un solveur (dit d'interface) sur la frontière des sous-domaines. Le code est donc bien parallèle, de la construction du système linéaire jusqu'à sa résolution (chaînage des parallélismes 1b+3a).
La distribution des sous-domaines[16] s'effectue par distribution cyclique. Lorsqu'on distribue q sous-domaines sur m processeurs : le sous-domaine 1 est attribué au processeur 0, le SD 2 au proc. 1, le SD q au proc. q, le q+1 au proc. 0, ...
Un paramètre du mot-clé SOLVEUR (NB_SD_PROC0=r[17]) permet de soulager la charge du processeur maître. En lui affectant moins de sous-domaines que ne le prévoit la distribution cyclique, il peut plus se consacrer (en temps et en mémoire) à la coordination du problème d'interface. Cette astuce peut être payante lorsque bon nombre de sous-domaines sont flottants. La gestion de leurs modes rigides peut alors tellement pénaliser le processeur 0, qu'il peut être intéressant de rééquilibrer artificiellement la charge en lui enlevant 1 ou 2 sous-domaine(s).

La mise en œuvre de ce parallélisme s'effectue de manière transparente à l'utilisateur. Elle s'initialise par défaut dès qu'on lance un calcul *via Astk* (menu Options) utilisant plusieurs processus MPI.
Ainsi sur le serveur centralisé *Aster*, il faut **paramétrer les champs suivants** :
- mpi_nbcpu=m, nombre de processeurs alloués en MPI (nombre de processus MPI).
- (facultatif) mpi_nbnoeud=p, nombre de nœuds sur lesquels vont être dispatchés ces processus MPI.

Cependant il faut veiller à ce que le paramétrage (m,q,r) reste licite : il faut au moins un SD par processeur (même pour le proc. 0).
Une fois ce nombre de processus MPI fixé, on peut lancer son calcul (en batch sur la machine centralisé) avec le même paramétrage qu'en séquentiel. On peut bien sûr baisser les spécifications en temps du calcul et surtout en mémoire.

15 Concernant les gains mémoire que procurer FETI, ceux-ci peuvent résulter de la conjugaison de l'effet multi-domaines, de l'OOC et de la distribution de données inhérente au parallélisme.
16 Préalablement construits *via* les opérateurs DEFI_PART*** et renseignés dans le mot-clé PARTITION.
17 Equivalent du CHARGE_PROC0_SD des opérateurs DEFI/MODI_MODELE.

Remarques:

- *Cette répartition sous-domaine/processeur s'effectue au début de chaque opérateur de calcul. On ne tient donc pas compte d'une éventuelle distribution de données par sous-domaines paramétrée en amont dans `AFFE/MODI_MODELE`.*
- *Empiriquement, on constate qu'il est intéressant pour gagner en mémoire, de découper le problème en de nombreux sous-domaines. Par contre, la résolution du problème d'interface et les déséquilibres de charges pénalisent alors les temps de résolution. Il est alors conseillé d'affecter plusieurs sous-domaines par processeur (équilibrage de charge empirique !). Le distinguo opéré dans Code_Aster entre sous-domaine et processeur procure cette liberté.*

7 Annexe 1 : Construction d'une version parallèle de *Code_Aster*

On peut mettre en oeuvre un calcul parallèle sur d'autres plate-formes que la machine *Aster* (cluster, noeuds de SMP...) voire sans l'usage d'*Astk*. Tout d'abord, il faut faire un choix entre le parallélisme MPI et celui OpenMP. Dans l'absolu, on peux cumuler les deux dans une même version dite parallèle hybride (c'est le cas sur la machine centralisée) mais, en première approche, on va ici les séparer.

Le wiki du site internet (http://www.code-aster.org/wiki) est une mine d'informations complémentaires (et actualisées) sur ces aspects informatiques.

7.1 Version parallèle OpenMP

Ce paragraphe concerne le scénario parallèle 2a (cf. §3).
Partant d'une version séquentielle du code et d'un compilateur OpenMP compatible (`Intel, gcc`...), il faut recompiler les sources après avoir modifié le fichier de configuration *Aster* (`config.txt`) :

- Rajouter l'**option appropriée pour le compilateur** (par ex. `icc -openmp`) *via* les listes d'options `OPTC_D/O, OPTF90_D/O`... Il y en a une pour chaque langage et chaque mode d'utilisation (debug ou optimisé).
- Faire la même chose au niveau du **linker** (ligne `OPTL` du `LINK`).
- Positionner l'**option de précompilation** `-D USE_OPENMP` pour le préprocesseur C (lignes `OPTC_D/O`).

Une fois la version OpenMP compilée, il faut transmettre le nombre de threads que l'on souhaite utiliser lors du calcul. Ce paramètre est transmis par *Astk* aux scripts de lancement (`.btc`, `exec_aster`) par l'intermédiaire de la variable d'environnement `OMP_NUM_THREADS`.

Remarque :
> *Attention à ne pas activer le parallélisme OpenMP des BLAS (cf. §4 .3). Ce dernier est contre-productif avec ce scénario parallèle.*

7.2 Version parallèle MPI

Ce paragraphe concerne les scénarios parallèles 1b, 2b, 2c et 3a (cf. §3).
À grands traits, il faut tout d'abord :

- **Choisir une implémentation MPI** compatible/optimisée pour sa plate-forme matérielle (machine, réseau, mémoire) et l'activer si nécessaire (`mpdboot` ...).
- Si l'on souhaite utiliser `MUMPS` ou `PETSC` en parallèle, les **installer/compiler avec leurs dépendances** : BLACS, Scalapack, METIS... (cf. wiki).
- **Recompiler les sources *Aster*** (*a minima* celles traitant du parallélisme) avec les «*wrappers*[18] MPI» et les options de pré-compilations *ad hoc* (`USE_MPI/MPI_MUMPS/FETI, HAVE_MUMPS/PETSC`...). Pour ce faire, on pourra s'inspirer du fichier de configuration *Aster* `config.mpi`, pendant du `config.txt` standard utilisé en séquentiel (cf. wiki).
- Il faut **préparer son calcul en mode séquentiel** (par ex. avec le menu '`pre`' d'*Astk*).
- Si on ne souhaite pas utiliser *Astk*, **il faut écrire un script shell** (cf. exemple ci-dessous) qui **duplique pour chaque processus MPI, l'espace de travail initialisé** par l'étape précédente sur chaque processeur alloué, qui **source l'environnement de travail** et qui **lance le calcul *Aster*** avec le paramétrage usuel. La syntaxe exacte de ces deux derniers points et le nom du répertoire de travail sont rappelés en fin de fichier message, à l'issu de la préparation du calcul.

18 En général une implémentation MPI se présente sous la forme de *wrappers* (par ex. `mpicc/mpif90`) et d'un lanceur. Le *wrapper* est un enrobage des compilateurs qui appellent les bons fichiers d'entête sur la ligne de compilation et fait l'édition de liens avec les bonnes bibliothèques MPI. Le lanceur (par ex. `mpiexec/mpirun/prun`) permet, comme son nom l'indique, de lancer l'exécutable parallèle.

```
mkdir -p /tmp/mon_etude_$$    (Nom aléatoire de la directory locale)
cp -r /home/mon_etude_preparee/* /tmp/mon_etude_$$    (Préparation de la directory de travail)
cd /tmp/mon_etude_$$    (On se place dans ce nouvel espace de travail)
export OMP_NUM_THREADS=1 (Pour être uniquement en MPI)
. /aster/ASTK/ASTK_SERV/conf/aster_profile.sh (On source l'environnement)
...
./aster.exe Eficas/Execution/E_SUPERV.py -eficas_path etc.    (Lancement effectif par le proc.)
rm -r /tmp/mon_etude_$$  (On n'oublie pas de faire le ménage !)
```

Figure A1.1._ Exemple de script shell pour préparer/lancer un calcul parallèle MPI.

- Il ne reste plus qu'à se **positionner sur la frontal et à utiliser le lanceur MPI**. Par exemple, *via* une commande du type `prun -n 16 nom_du_script.sh`.